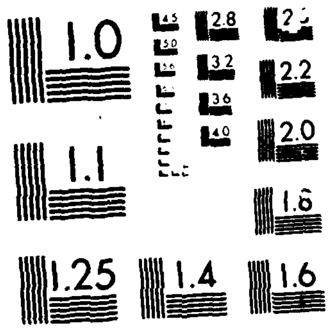
AD-A166 058 ON USING MULTIPLE INVERTED TREES FOR PARALLEL UPDATING OF GRAPH PROPERTIE..(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE S PAWAGI ET AL. MAY 85 CAR-TR-124 1/1

UNCLASSIFIED AFOSR-TR-86-0045 N00014-84-K-0530 F/G 12/1 NL

END
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART

CAR-TR-124
CS-TR-1502

May 1985

On Using Multiple Inverted Trees for Parallel Updating of Graph Properties

Shaunak Pawagi
I. V. Ramakrishnan
Department of computer Science
University of Maryland
College Park, MD 20742

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

CAR-TR-124
CS-TR-1502

May 1985

On Using Multiple Inverted Trees for Parallel Updating of Graph Properties

Shaunak Pawagi
I. V. Ramakrishnan
Department of computer Science
University of Maryland
College Park, MD 20742

[illegible]

ABSTRACT

Fast parallel algorithms are presented for updating the distance matrix, shortest paths for all pairs and biconnected components for an undirected graph and the topological ordering of vertices of a directed acyclic graph when an incremental change has been made to the graph. The kinds of changes that are considered here include insertion of a vertex or insertion and deletion of an edge or a change in the weight of an edge. The machine model used is a parallel random access machine which allows simultaneous reads but prohibits simultaneous writes into the same memory location. The algorithms described in this paper require O(log n) time and use $O(n^3)$ processors. These algorithms are efficient when compared to previously known $O(\log^2 n)$ time start-over algorithms for initial computation of the above mentioned properties of graphs. The previous solution is maintained in multiple inverted trees (a rooted tree where a child node points towards its parent) and after a minor change the new solution is rapidly recomputed from these trees.

+ The support of the first author by the Air Force Office of Scientific Research under Contract F-49620-83-C-0082, and of the second author by the Office of Naval Research under Contract N00014-84-K-0530, and by the National Science Foundation under grant ECS-84-04399, is gratefully acknowledged.

1. Introduction

Incremental graph algorithms deal with recomputing properties of graph after an incremental change has been made to the graph. Such recomputations are also referred to as "updating" graph properties. Sequential incremental algorithms for recomputing minimum spanning trees [8] connected components [3] and shortest paths [4] have appeared in the past. However, except in [7], incremental algorithms have not been studied for any model of parallel computation. Parallel algorithms for several graph problems have been devised [5,6,8,9 and 11] for an unbounded model of parallel random access machine (PRAM). In this model of computation all processors have access to a global memory and processors can simultaneously read from the same location but no two processors can simultaneously write into the same location. The algorithms developed on this model of computation provide us a basis to compare the complexity of our incremental algorithms. In this paper we describe incremental algorithms for updating the distance matrix, shortest paths for all pairs and biconnected components of an undirected graph and topological ordering of the vertices of a directed acyclic graph (DAG). Our algorithms for updating these properties require O(log n) time on a PRAM and therefore are efficient when compared to the start-over algorithms for initial computation of these properties that require $O(\log^2 n)$ time.

The kinds of minor modifications that are considered here are as follows. First, a vertex may be added along with the edges incident on it. Second, an individual edge may be deleted or added. If an edge has a weight associated with it then an increase or decrease in its weight is permitted. Note that an increase or decrease in the weight of an edge encompasses an edge deletion or insertion operation, because we can treat the nonexistent edges as having infinite weights.

An important aspect of incremental algorithms is the design of data structures to store the previous solution as well as some auxiliary information that is generated during the initial computation Such data structures should provide rapid access to the necessary information for efficient updates of the solution. As we will see later on our update algorithms require fast identification of the vertices that belong to two different subtrees that are created by deleting an edge from the tree. For an inverted tree (a rooted tree where a root node points towards its parent) this computation can be done in O(log n) time (see [11]). It was shown in [7] that an inverted spanning tree can be used to update a minimum spanning tree, connected components and bridges of an undirected graph in O(log n) time on a PRAM. In this paper we use n such inverted spanning trees for updating the distance matrix, shortest paths for all pairs, biconnected components and topological ordering of the vertices. Specifically, in the distance matrix the i^{th} row defines a distance tree (DT) for the underlying graph and such a tree is rooted at vertex i. (A distance tree is in fact a shortest path tree for an unweighted graph.) Now, for a graph ·ith n vertices, if we store n such trees (each having a different root) then we can quickly identify the pairs of vertices whose distances must be recomputed after the graph undergoes a minor change. In the case of shortest paths for all pairs, we store the shortest path trees as inverted trees. The longest path trees that are involved in the computation of topological ordering of the vertices of a DAG, are stored as inverted trees. In order to update biconnected components efficiently we store n inverted spanning trees, one for each graph G_i that is obtained from the original graph G by deleting vertex i.

The rest of the paper is organized as follows. In Section 2 we describe some graph-theoretic preliminaries adopting the framework in [11]. In Section 3 w· describe our algorithms for updating the distance matrix. In Sections 4 and 5 we extend the ideas of

Section 3 to update shortest paths for all pairs and topological ordering of the vertices of a DAG respectively. Our algorithms for updating biconnected components are described in Section 6.

2. Preliminaries

In order to describe our algorithms to update graph properties we now present some graph theoretic preliminaries.

Let G=(V,E) denote a *graph* where V is a finite set of vertices and E is a set of pairs of vertices called edges. If the edges are unordered pairs then G is *undirected* else it is *directed*. Throughout this paper we assume that $V=\{1,2,...,n\}$, $|V|=n$ and $|E|=m$. We denote the undirected edge from a to b by (a,b) and the directed edge between them by $<a,b>$. We say that an undirected graph G is *connected* if for every pair of vertices u and v in V, there is a path in G joining u and v. Each connected maximal subgraph of G is called a *component* of G. An adjacency matrix A of G is an $n \times n$ Boolean matrix such that $A[u,v]=1$ if and only if $(u,v) \in E$. A *tree* is a connected undirected graph with no cycles in it. Let $T=(V', E')$ be a directed graph. T is said to have a *root* r, if $r \in V'$ and every vertex $v \in V'$ is reachable from r via a directed path. If the underlying undirected graph of T is a tree then T is called a directed tree. If the edges of T are all reversed then the resulting graph is called an *inverted* tree. An *inverted spanning tree* (IST) and an *inverted spanning forest* (ISF) are defined similarly. We denote an undirected path from vertex a to vertex b by [a-b] and directed path by $[a \rightarrow b]$. We say that vertex w is an ancestor of vertex v if w is on the path from v to the root of the tree. Let T be a directed tree with $u,v \in V'$. Then the *lowest common ancestor* (LCA(u,v)) of u and v in T is the vertex $w \in V'$ such that w is a common ancestor of u and v, and any other common ancestor of u and v in T is also an ancestor of w in T.

As we will see later on, our update algorithms require the paths from all vertices to the root in an inverted tree. Tsin and Chin [11] have described a technique due to Savage [8] to compute all such paths. For completeness we now describe their technique.

Let $T=(V',E')$ be an inverted tree with $V'=\{1,2,...,n\}$ and $|V'|=n$. Let r be the root of this tree. For a directed edge $<a,b>$ we say that vertex b is the father of vertex a.

Definition: $F:V' \rightarrow V'$ is a function such that F(i)=the father of vertex i in T for $i \neq r$ and $F(r)=r$.

The function F can be represented by a directed graph F which can be constructed from T by adding a self-loop to the root r.

From the function F, we define F^k, $k \geq 0$ as follows.

Definition: $F^k:V' \rightarrow V'$ $(k \geq 0)$ such that $F^0(i)=i$ for all $i \in V'$ and $F^k(i)=F(F^{k-1}(i))$ for all $i \in V'$ and $k>0$.

If i is a vertex in T, $F^k(i)$ is the k^{th} ancestor of i in the inverted tree.

Definition: For each $i \in V'$, $depth(i)=\min\{k|F^k(i)=r \text{ and } 0 \leq k < n\}$.

Lemma 2.1: Given the function F of an inverted tree, F^k can be computed in O(log n) ++ time using $O(n^2)$ processors.

Proof: To compute F^k $(0 \leq k < n)$ we proceed as follows. We assume that the processors are indexed as P(1,1), P(1,2),..., P(n,n). The instructions within "pardo...dopar" are executed in parallel and comments are enclosed within //...//.

++Throughout this paper, we use log n to denote $\lceil \log_2 n \rceil$

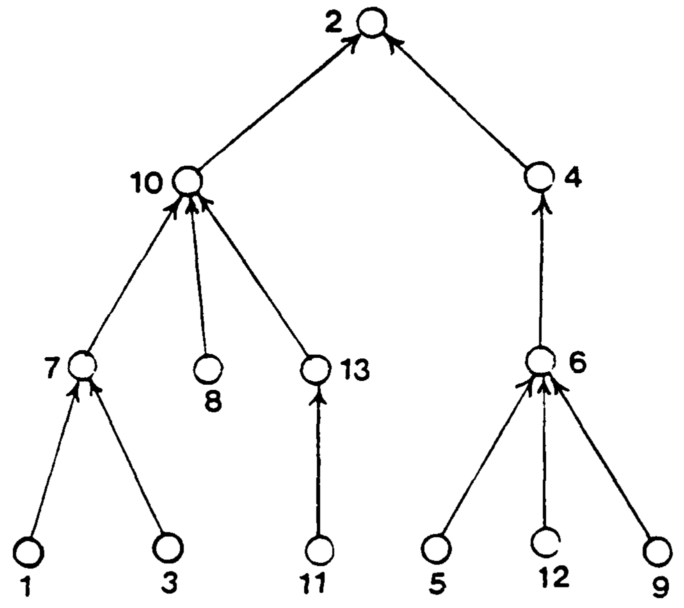
```
//Processor P(1,i) executes the instruction within pardo...dopar.//
1.  for all i ($1 \leq i \leq n$) pardo $F^0(i)=i$, $F^1(i)=F(i)$ dopar;
2.  for t:=0 to log(n-1)-1 do
    //Processor P(s,i) executes the instruction within pardo...dopar.//
    for all s ($1 \leq s \leq 2^t$) and for all i ($1 \leq i \leq n$) pardo $F^{2^t+s} := F^{2^t}(F^s(i))$ dopar;
```

We compute the function F^k by successive composition of the functions F^i $i < k$, which have been determined in previous iterations. The number of iterations that are marked in each iteration increase by a factor of two. Since a vertex can have at most n-1 ancestors, we need O(log n) iterations to mark all ancestors. Now step (1) can be done in constant time using n processors. To do the i^{th} iteration of step (2) in constant time we require $2^i n$ processors. As there are log(n-1)-1 iterations of step (2), we therefore require $O(n^2)$ processors. $\square$

The actual computations of $F^k(i)$ ($1 \leq i \leq n$, $1 \leq k < n$) are performed in an array F^+ in which $F^+[i,k]$ contains $F^k(i)$. Once the F^+ array is computed, depth(i) ($1 \leq i \leq n$) can be found by performing a binary search on the i^{th} row. We search for the leftmost occurrence of r. This takes log n time by assigning a processor per row. However, it can be done in constant time by assigning a processor to each element in F^+. This is done as follows. Every processor compares its element with the elements in its left and right neighbors. There is exactly one processor which does not have all the three elements identical or distinct and this processor locates the leftmost occurrence of r. The depth information is stored in a one-dimensional array H^+.

After the computations for H^+ are finished, each row of F^+ is right shifted so that all the r's except the leftmost one are eliminated. As a consequence, the rightmost column of the array contains only the root r. Fig. 2.1 illustrates an inverted tree and its array F^+ after the rows have been shifted right.

Lemma 2.2: We can compute the lowest common ancestors of all vertex pairs in the inverted tree in O(log n) time using O(n^2) processors.



	0	1	2	3	4	5	6	7	8	9	10	11	12
1										1	7	10	2
2													2
3										3	7	10	2
4												4	2
5										5	6	4	2
6											6	4	2
7											7	10	2
8											8	10	2
9										9	6	4	2
10												10	2
11										11	13	10	2
12										12	6	4	2
13											13	10	2

Undefined entries are left blank.

Fig. 2.1

Proof: We make use of the array F^+ to design a parallel algorithm for finding the lowest common ancestors. For an n vertex graph there are ${}^nC_2$ (the number of unordered pairs of n elements) vertex pairs, that is $O(n^2)$ pairs. Let a and b be a vertex pair. If c is their lowest common ancestor, then row a and row b of F^+ will have identical contents for column n-1, column n-2,..., down to the column containing c. After this column the contents of rows a and b differ. As a result, to determine c, we can perform a binary search on row a and row b simultaneously in the following way. If the two entries being examined in row a and row b (in the same column) are different, the search is continued on the right half, otherwise it is continued on the left half. It takes (log n)+1 time steps to find c with one processor. Therefore we need $O(n^2)$ processors to find the lowest common ancestors of all vertex pairs. $\square$

Having obtained the lowest common ancestor we can now identify the unique path between any two vertices (passing through their lowest common ancestor).

3. Distance Matrix

In this section we consider updating the distance matrix of an undirected graph whose edges have uniform weights. For a graph G its distance matrix D is defined as follows.

$$D[i,j] = \begin{cases} 1, & \text{if (i,j) is an edge of G} \\ d, & \text{d is the length of the shortest path [i-j]} \\ \infty, & \text{if no such path exists} \end{cases}$$

Note that D is symmetric and the adjacency matrix A is contained in D. The distance matrix has several applications. For instance, the radius, diameter and the center of a graph are defined in terms of its distance matrix. The distance matrix has also been used in for detecting graph isomorphism.

The problem of updating the distance matrix involves recomputing D for the modified graph that is obtained from the original graph after a minor change has been made to it. Cheston [1] has studied sequential algorithms for updating the distance matrix. Our algorithms for updating the distance matrix on a PRAM require O(log n) time and use $O(n^3)$ processors. The start-over algorithm for initial computation of D requires $O(\log^2 n)$ time and use $O(n^3)$ processors. Our algorithms therefore are efficient when compared to the start-over algorithm.

To design efficient parallel algorithms for updating the distance matrix we proceed as follows. The first step is to determine the vertex pairs whose distances are unaffected by the graph change. In particular, we need to compute these pairs after an edge has been deleted from G. Note that the i^{th} row of D corresponds to a DT (distance tree) for G, that is rooted at vertex i. Our update algorithms require that n such DTs for G, one for each row of D, be stored along with the matrix D. We speed up the computation by storing these trees as inverted trees. Now, an edge deletion operation may create two subtrees out of a single DT. Using the techniques described earlier (see Section 2), we can identify the vertices that belong to each of these subtrees in O(log n) time. Therefore we can determine the entries of D that are unaffected by the graph change. After the matrix D has been recomputed we can reconstruct these DTs as inverted trees.

The other cases of edge and vertex insertion can be handled without DTs. We do not consider the problem of vertex deletion, since deletion of a vertex from a tree may split it into more than two subtrees. We are unable to handle this situation using our approach.

In order to describe the actual computational steps of our algorithms and the proof of their correctness, we first describe the parallel start-over algorithm for computation of the distance matrix.

Start-Over Algorithm

It has been observed in [2] that distances for all pairs of vertices in a graph (directed or undirected) can be computed in O($\log^2 n$) time on a PRAM by straightforward parallelization of the known sequential algorithm that is based on repeated multiplication of the adjacency matrix. In this parallel algorithm addition and minimization operations replace multiplication and addition operations of an inner product step. We refer to this as the *plus-min* multiplication of two matrices. The algorithm initializes the distance matrix D to the adjacency matrix A and then performs (log n) iterations of the plus-min multiplication of D by itself. The matrix DD is used as temporary storage in the algorithm for clarity.

```
// All steps involving i and j are executed for all i, j $1 \le i \le n$ and $1 \le j \le n$ //
1.   D[i,j] := A[i,j]     //Initialize//
2.   for t:=1 to log(n-1) do
2a.      DD[i,j] := min { D[i,j], D[i,k] + D[k,j] } // $1 \le k \le n$ $i \ne k$ $j \ne k$//
2b.      D[i,j] := DD[i,j]
```

Algorithm 3.1

Lemma 3.1: The above algorithm computes the distance matrix D in O($\log^2 n$) time using O(n^3) processors.

Proof: Steps (1) and (2b) can be done in constant time using n^2 processors. Step (2a) can be done in O(log n) time by assigning n processors to compute each element of the matrix DD Since DD has n^2 elements we need O(n^3) processors to perform step (2a). Note that at the end of t^{th} iteration we would have found distances for those pairs whose vertices are at most 2^t units apart. Since the maximum distance for any pair of vertices is at most (n-1) units, we need log(n-1) iterations of step (2a). □

Construction of Distance Trees from D

We now describe the computational steps for constructing n DTs, each rooted at a different vertex. Let T_i denote a DT that is rooted at vertex i. Recall that the function F completely specifies an inverted tree (see Section 2). Let the function F_i completely specify T_i. F_i is computed as follows.

1. Set $F_i(i) = i$, because i is the root of the tree.
2. For every vertex in T_i other than its root, determine its father in T_i. This can be done in O(log n) time using n processors for a vertex. Consider vertex j and let its distance from the root be d_j. Therefore the father of j must be a vertex that is $(d_j$-1) units away from the root. Assign n processors to j and select a vertex k such that k is adjacent to j and $(d_j$-1) units away from i. Select a minimum k to break the ties. Set $F_i(j) = k$. Since j can have at most n neighbors this minimization needs O(log n) time and n processors. As there are n vertices in each tree we need $O(n^2)$ processors to construct T_i. Therefore we have the following lemma.

Lemma 3.2 The construction of n DTs from the matrix D requires O(log n) time and uses $O(n^3)$ processors.

We now proceed to describe our update algorithms and the proof of their correctness.

Edge deletion

The problem of edge deletion update is concerned with recomputing the distance matrix D after an edge has been deleted from the graph. In order to recompute D, we first identify the pairs of vertices whose distances are unaffected by the edge deletion step. We then construct matrix D′ such that

$$D'[i,j] = \begin{cases} D[i,j], & \text{if } D[i,j] \text{ is unchanged.} \\ \infty & \text{otherwise} \end{cases}$$

Now, two iterations of steps (2a) and (2b) of Algorithm 3.1 on D′ recompute the distance matrix for the new graph. We will show later on that two iterations are sufficient for recomputing D. The computational steps are described for tree T_i. There are n such trees and the following steps for T_i are executed in parallel for all of them.

Let (x,y) be the edge that was deleted from G. Also assume that the T_i's have been constructed from D.

1. If (x,y) is not in T_i then the i^{th} row of D is not affected at all. This is so because T_i stays connected even after (x,y) has been deleted from G. Therefore the distances to all other vertices from i in the new graph do not change.

   Now suppose that the edge (x,y) is in T_i. Assume, without loss of generality that its direction in T_i is from x to y. To delete (x,y) set $F_i(x) = x$. This creates a forest of two subtrees one of which is rooted at i and the other at x.

2. Compute the array F_i^+ for T_i. The last column of this array identifies the vertices that belong to different subtrees. The vertices in the subtree rooted at x are not reachable from i in T_i. Therefore their distances from i must be recomputed. By Lemma 2.1, computation of the array F_i^+ takes O(log n) time and requires $O(n^2)$ processors.

3. Compute the i^{th} row of D′ as follows. The distances to the vertices that are not reachable from i are marked as ∞. For other vertices that are reachable from i, the distances have not changed. This step can be done in constant time using $O(n^2)$

processors.

4. Perform two iterations of the start-over algorithm on D′ to compute the updated distance matrix. This computation requires O(log n) time and uses $O(n^3)$ processors

We now prove the correctness of our algorithm.

Theorem 3.1: Two iterations of the start-over algorithm that operates on D′ are sufficient to compute the updated distance matrix for the new graph.

Proof: Consider the tree rooted at i (see Fig. 3.1). Let (x,y) be the edge that was deleted. If (x,y) is a bridge of G then (x,y) is present in all DTs, and hence in T_i. Deletion of (x,y) disconnects G as well as all T_i's. The vertices in the subtree rooted at x are not reachable from i in G. Therefore D′ itself is the new distance matrix.

On the other hand if (x,y) is not a bridge of G then there exists at least one edge in G that connects these two subtrees that were created by deletion of (x,y). Let u be a vertex in the subtree rooted at x that has such an edge incident on it. Since T_i is a DT for G, u can have neighbors that are at a distance d_u, d_u+1 or d_u-1 (where d_u is the distance to u from i). Therefore at the end of the first iteration (i.e., *plus-min* multiplication of D′ by itself) we would have found the distances to all such u's from i.

Now consider a vertex v that does not have a neighbor in the subtree rooted at i. The shortest path from i to v must pass through some such u (a vertex that has a neighbor in the subtree rooted at i). Therefore the distance to v from i can be expressed as the sum of distances, one from i to u and the other from u to v. Now, at the end of the second iteration we would have computed the distances to all such v's from i. This requires that D′ [u,v] must not have been marked as ∞ in step (3) of our algorithm. In other words, v must be reachable from u in the DT T_u even after the edge (x,y) has been

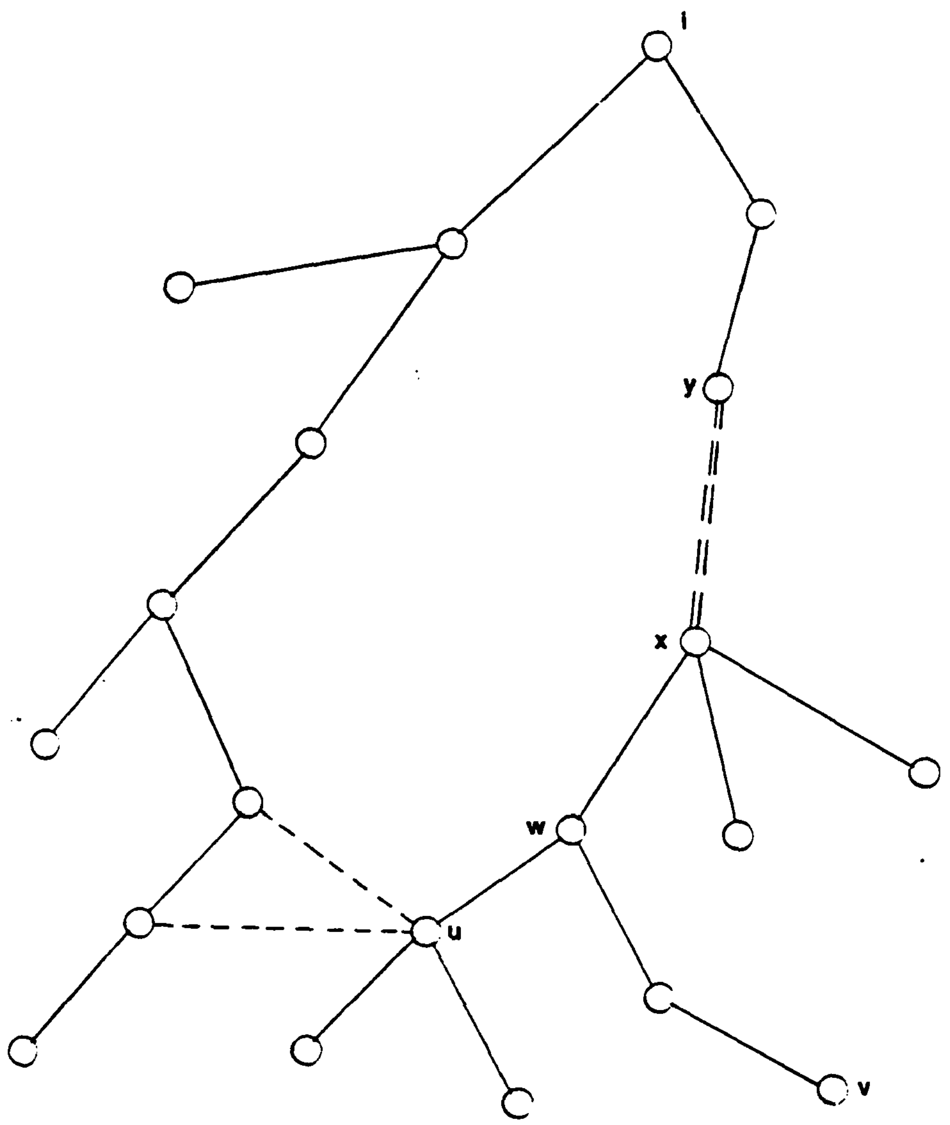


Fig. 3.1

deleted from G. We now show that this indeed is the case !

The subtree rooted at x contains a path [u-v] that passes through their lowest common ancestor. The length of this path is $(d_u-d_v-2d_w)$, where $w = LCA(u,v)$. This path is shorter than any other path [u,v] that contains the edge (x,y), because the length of such a path is at least $(d_u-d_v-2d_y)$ and $d_y<d_w$ (note that y is an ancestor of w).

Therefore the DT T_u contains the shortest path from u to v that is either a concatenation of paths [u-w] and [w-v], or a path shorter than this, which does not contain (x,y). Hence the distance D[u,v] is not affected by deletion of (x,y).

It is possible that vertex v is reachable from many such u's, but the minimization operation will select a vertex that minimizes the length of the path from i to v. Therefore at the end of the second iteration we would have computed the shortest paths to all vertices from the root i. Hence the theorem. $\square$

The ideas described in this proof will be used in subsequent sections to outline the similar proofs.

Theorem 3.2: Our algorithm updates the distance matrix of an undirected graph after an edgè deletion operation in O(log n) time and use $O(n^3)$ processors.

Proof: By Lemma 3.2, the construction of the DTs needs O(log n) time and $O(n^3)$ processors. The computation of the array F_i^+ requires O(log n) time and $O(n^2)$ processors. As there are n such trees, we need $O(n^3)$ processors for step (2). The matrix D′ can be constructed from D and the arrays F_i^+ in constant time using $O(n^2)$ processors. Now two iterations of the start-over algorithm are sufficient to compute the new distance matrix. By Lemma 3.1, each iteration needs O(log n) time and uses $O(n^3)$ processors. $\square$

Edge and Vertex Insertion

We now describe our algorithms for updating the distance matrix after an edge or a vertex has been inserted into G. In order to compute D′ from D after an edge insertion operation we proceed as follows. Let (u,v) be the edge that has been inserted into G.

1. Set D′ [u,v] = D′ [v,u] = 1. All other entries of D′ are the same as that of D.

2. Perform two iterations of the start-over algorithm that uses D' as its input to compute the updated distance matrix.

In the case of vertex insertion we add a new row and a column to the old distance matrix. Let z be the new vertex that has been inserted into G. Now D' can be obtained from D by setting $D'[z,w] = D'[w,z] = 1$, for all w, where w is adjacent to z. All the other entries in the z^{th} row and in the z^{th} column are marked as ∞. Again, two iterations of the start-over algorithm recompute the new distance matrix.

Theorem 3.3: Our algorithms for edge and vertex insertion update require O(log n) time and use $O(n^3)$ processors.

Proof: For an edge insertion update we can compute D' from D in constant time using one processor and this step for a vertex insertion update requires 2n processors. The rest of this proof is along lines similar to that of Theorem 3.1. $\square$

4. Shortest Paths

We now extend the techniques of the previous section to handle weighted graphs. The problem of updating the distance matrix gets transformed to updating shortest paths in an undirected graph. Let $L:E \rightarrow R$ denote a function that associates a length with the edges of G. The lengths of the shortest paths for all pairs are stored in a matrix P such that $P[i,j]$ is the length of the shortest path from vertex i to vertex j.

The problem of updating shortest paths involves recomputing the shortest path matrix P from the previous such matrix when the length of an edge has changed or a vertex has been inserted or deleted from the graph. We refer to these two problems as the edge update and the vertex update problem respectively. The start-over algorithm in [2] for computing shortest paths for all pairs on a PRAM requires $O(\log^2 n)$ time and uses $O(n^3)$ processors. Our algorithms for the above mentioned update problems require $O(\log n)$ time and use $O(n^3)$ processors.

Recall that our algorithms for updating distance matrix use n DTs of the graph. For updating shortest paths we employ shortest path trees (SPTs) for this purpose. Since we are dealing with weighted graphs, the shortest path matrix P cannot be used to construct the SPTs. In order to construct SPTs using the technique for constructing the DTs, we actually need the number of edges on all shortest paths. Therefore, the start-over algorithm described in the previous section needs to be modified to compute this. This information is stored in the matrix D wherein D[i,j] is the number of edges on the shortest path from i to j. The SPTs that are constructed from D are maintained as inverted shortest path trees.

We now describe the start-over algorithm for computing the shortest paths that has been modified to compute D.

Start-Over Algorithm

As described earlier, the parallel algorithm for computing shortest paths for all pairs is based on the repeated plus-min multiplication of matrix P that has been initialized to edge-lengths. The computational steps are as follows.

```
    //All steps involving i and j are executed for all i,j   $1 \leq i \leq n$   $1 \leq j \leq n$//
1.  P[i,j] := L(i,j)    //the length of edge (i,j)//
    D[i,j] := A[i,j]    //A is the adjacency matrix.//
2.  for t := 1 to log(n-1) do
2a.     PP[i,j] := min { P[i,j], P[i,k] + P[k,j] }  //k $\neq$ i and k $\neq$ j//
2b.     DD[i,j] := D[i,k] + D[k,j]   //for the same k obtained in the step 2a//
2c.     P[i,j] := PP[i,j];  D[i,j] := DD[i,j]
```

Algorithm 4.1

Lemma 4.1: The above algorithm computes the shortest path matrix P in $O(\log^2 n)$ time and requires $O(n^3)$ processors.

Proof: The proof of this lemma is similar to that of Lemma 3.1. □

We now proceed to describe our algorithms for updating shortest paths. First, we compute the matrices D′ and P′ that contain the information unaffected by the incremental change. Now two iterations of the start-over algorithm on D′ and P′ are sufficient to recompute D and P for the new graph.

Edge Update

Assume that the length of an edge has changed. There are several cases to be handled. First, the length of an edge may either decrease or increase and this edge may either currently be in some of the SPTs or may not be in any of the SPTs. We describe our algorithm with respect to a SPT T_i that is rooted at i. Note, however, that there are n such trees and all of them are processed in parallel.

Let (x,y) be the edge whose length has decreased. Assume, without loss of generality that if (x,y) is present in T_i then its direction is from x to y. If (x,y) is not in T_i then let y be closer to the root i than x.

1. If (x,y) is not in T_i then there must exist a shortest path [x-y] such that $L(x,y) \geq P[x,y]$. Now if the new length of (x,y) is still greater than or equal to that of the path [x-y] then T_i does not change. Therefore the i^th row of D and P remains unchanged.

   On the other hand if the new $L(x,y) < P[x,y]$ then we have a shorter path from i to x and to all descendants of x (see Fig. 4.1). Let $\delta = D[x,y] - 1$. Now the number of edges on the shortest paths from i to x and to descendants of x have reduced by

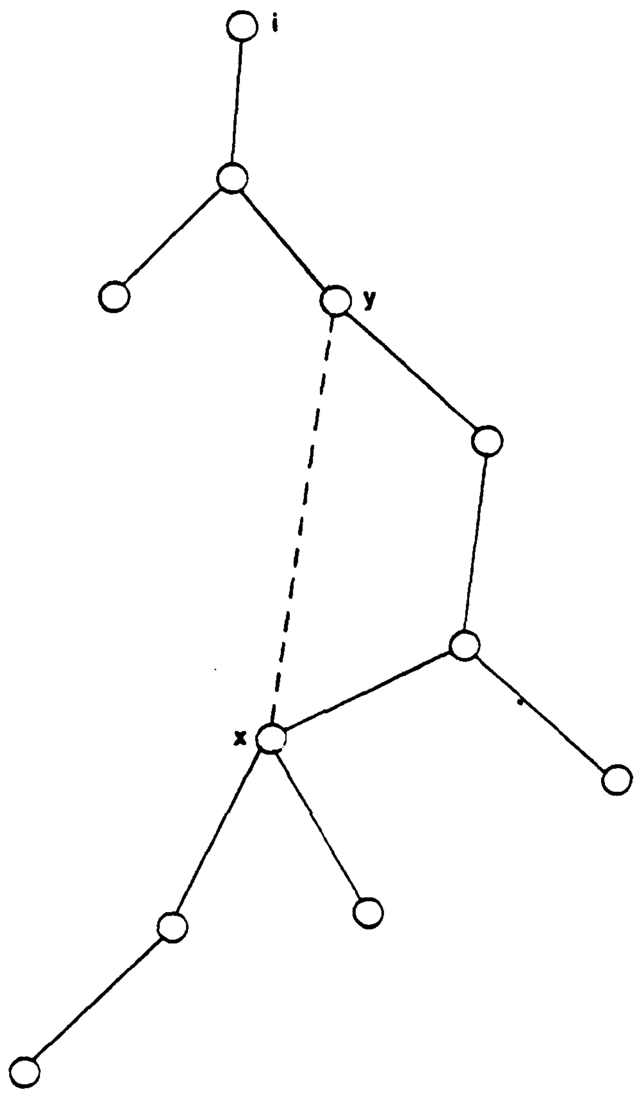


Fig. 4.1

δ. Therefore set $D'[i,j] = D[i,j] - \delta$, where j is either x or a descendant of x. The remaining entries of the ith row of D' are the same as that of ith row of D.

Similarly, obtain the ith row of P' by setting $P'[i,j] = P[i,j] - \theta$, where j is either x or a descendant of x and $\theta = P[x,y] - L(x,y)$. Here, the lengths of the shortest paths from i to x and to its descendants have reduced by θ. The other entries of the ith row of P' are the same as that of the ith row of P.

2. If (x,y) is in T_i then the shortest paths from i to x and to its descendants have reduced by Δl, where Δl is the reduction in the length of (x,y). In this case D remains unchanged but the ith row of P is affected. Therefore set $P'[i,j] = P[i,j] - \Delta l$, where j is either x or a descendant of x.

3. Perform one iteration of the start-over algorithm to recompute the matrices D and P for the new graph.

Now suppose that the length of (x,y) has increased. If (x,y) is not in T_i then none of the distances from i to other vertices change. Hence the ith row of D and P are unaffected. If (x,y) is in T_i then set $P'[x,y]$ to the new length of (x,y) and the rest of P' is the same as P. The matrix D is not affected at all. In this case two iterations of the start-over algorithm on D' and P' update the matrices D and P.

Theorem 4.1: Our algorithm for the edge update problem requires O(log n) time and uses $O(n^3)$ processors.

Proof: We can find descendants of x in constant time using our computation of the function LCA (see Section 2). For instance, vertex w is a descendant of x if LCA(x,w) is x. We need n-1 processors to determine all descendants of x in constant time. By Lemma 2.2, LCA for all pairs of vertices can be obtained in O(log n) time using $O(n^2)$ processors. All other steps except the iterations of the start-over algorithm can be done in constant time using $O(n^2)$ processors. As there are n such SPTs we need $O(n^3)$ processors. By Lemma 4.1, one iteration of the start-over algorithm can be done in O(log n) time using $O(n^3)$ processors. The proof of the sufficiency of two iterations for recomputing D and P is along the same line as that of Theorem 3.1. $\square$

Note that our algorithm can easily handle the cases where an edge is inserted or deleted from the graph, since we can treat the nonexistent edges as edges having infinite weights.

Vertex Update

The problem of vertex update deals with recomputing the shortest paths after a vertex has been inserted into the graph. We add a new row and a column to D' and P' for the new vertex z. Now set $D'[w,z] = D'[z,w] = 1$ and $P'[w,z] = P'[z,w] = L(z,w)$, where w is a neighbor of z. All other entries of z^{th} row and z^{th} column of D' and P' are respectively set to 0 and ∞. Now execute the start-over algorithm on D' and P'. Two iterations of step (4) would recompute the updated D and P. Therefore we have the following theorem.

Theorem 4.2: Our algorithm for the vertex update problem requires O(log n) time and uses (n^3) processors.

Proof: The proof of this theorem is similar to that of Theorem 3.3. $\square$

5. Topological Ordering

In this section we describe our algorithms to update the topological ordering of the vertices of a DAG after an incremental change has been made to G. This is an important property of DAGs and finds applications in activity networks and in critical path analysis of networks.

Definition: For different vertices u and v of a DAG G, u is a predecessor of v iff there is a directed path from u to v.

The topological ordering of the vertices have the property that if vertex i in G is a predecessor of vertex j, then i precedes j in the ordering of the vertices of G. The topological ordering is defined for directed acyclic graphs. Now, acyclicity of the graph may not be preserved after the graph has been modified. But our algorithms are capable of detecting such anomalies.

Dekel et al. [2] observed that the algorithm for shortest paths for all pairs can be used to compute the lengths of the longest paths. This requires using maximization in place of minimization on the shortest path algorithm. For ease of exposition, we now describe their algorithm. Note that we are dealing with an unweighted graph.

1. First, determine the lengths of the longest paths for all pairs using the technique discussed above. Store the results in the matrix D. This step needs $O(\log^2 n)$ time and $O(n^3)$ processors. Note that D is asymmetric and if $D[i,i] > 0$ for any i $1 \leq i \leq$, then G is not acyclic.

2. Set $D[1,i] = 0$ for all i such that $D[t,i] = 0$, for $1 \leq t \leq n$. Note now that $D[1,i] = 0$ for exactly those vertices in G that have no predecessors. This step can be done in $O(\log n)$ time using $O(n^2)$ processors by assigning n processors to a vertex. Let i_1, i_2, ..i_k be k such vertices that have no predecessors.

3. Now for each j such that $D[1,j] \neq 0$ (i.e. those vertices with predecessors) set $D[1,i] = \max \{ D[i_p,k] \}$ $1 \leq p \leq n$. Since this step involves finding a maximum of at most n numbers we need $O(\log n)$ time and $O(n^2)$ processors.

4. Sort the pairs $\{ D[1,j], j \}$ in increasing order to obtain the topological order of the vertices. Sorting requires $O(\log n)$ time and $O(n^2)$ processors.

Observe that except step (1) all other steps can be done in $O(\log n)$ time. Therefore, if we can recompute the matrix D for the modified graph in $O(\log n)$ time, then we have

O(log n) time algorithm for updating the topological ordering of the vertices of a DAG !

The i^{th} row of D defines a longest path directed tree that is rooted at i. The edges of these trees are oriented from root to the leaves. However, for computational purpose we store these trees as inverted trees. Using the techniques described in the earlier sections we construct the matrix D' from D in O(log n) time using $O(n^3)$ processors. Now two iterations of the longest path algorithm would recompute D. Perform the steps (2), (3) and (4) to obtain the topological order of the vertices of the new graph. The longest path trees can be constructed from D in O(log n) time using $O(n^3)$ processors, according to the technique for constructing DTs from D (see Section 3). Therefore we have the following theorem which is stated without proof.

Theorem 5.1: The topological ordering of the vertices of a DAG can be updated in *O(log n) time using $O(n^3)$ processors.*

The ideas employed here are essentially the same as those described in Sections 3 and 4. Therefore we have omitted the actual computational steps.

6. Biconnected Components

The problem of updating biconnected components of a graph deals with recomputing the sets of vertices, one for each component, after an incremental change has been made to the graph. The best known algorithm for finding biconnected components [11] requires $O(\log^2 n)$ time. Our approach to updating biconnected components is based on the start-over algorithm due to Savage and Ja'Ja'[9]. Our algorithms for edge and vertex insertion update require O(log n) time and use $O(n^3)$ processors.

We assume that the update algorithms operate on a set of inverted spanning forests (ISF) that is defined as follows. Let G_k denote a graph that is obtained from G by deleting vertex k. Deletion of k may split G into components, each of which can be represented by an inverted spanning tree for that component. Let S_i denote the corresponding ISF for G_k. To compute the biconnected components of G from this set of S_k's, we proceed as follows.

Let R be a binary relation on V. For $i,j \in V$, $i\ R\ j$ if i and j are in the same biconnected component of G. Now, i and j are in the same biconnected component if there does not exist vertex w, such that after removal of w from G, i and j are not reachable from each other in G_w. Therefore R can be determined from the G_k's as follows. $i\ R\ j$ iff for all k distinct from i,j they are in the same connected component of G_k. We can compute the relation R in O(log n) time by assigning n-2 processors to each vertex pair. First, we check in constant time if i and j are in the same connected component of G_k. The results of (n-2) such tests are then merged in a "binary tree" fashion to reach the final decision in O(log n) time. As there are $O(n^2)$ vertex pairs, to determine R we need $O(n^3)$ processors. In order to compute the biconnected components from R we use the following lemma.

Lemma 6.1: Let T be a spanning tree for G and (i,j) be an edge of T; then

$$V_{i,j} = \{ k \in V \,/\, i\ R\ k \text{ .and } j\ R\ k \}$$

is the vertex set of the biconnected component that contains the edge (i,j).

Proof: The proof is immediate from the definitions of R and the set $V_{i,j}$. □

Lemma 6.1 provides us an algorithm to determine the vertex sets of all biconnected components using an IST T for G and the relation R. We can construct all sets $V_{i,j}$ in constant time by assigning n processors to each edge of T. As there are n-1 edges in T we need $O(n^2)$ processors. Therefore we have the following theorem.

Theorem 6.1: The biconnected components of an undirected graph can be updated in O(log n) time using O(n^3) processors.

Proof: The ISF for a graph can be updated in O(log n) time using O(n^2) processors (see [7] for a proof). Therefore the collection of n ISFs, (i.e, S_k's), one for each G_k and an IST T for G can be updated in O(log n) time using O(n^3) processors. The computation of R from this collection needs O(log n) time and O(n^3) processors. The construction of vertex sets for biconnected components from the relation R can be done in constant time using O(n^3) processors. □

The sets of vertices thus constructed may have duplicates. For instance, if two edges of T, say (i,j) and (u,v), are in the same biconnected component then the corresponding vertex sets $V_{i,j}$ and $V_{u,v}$ that are determined using them are equal. It is easy to discard the multiple copies of such sets in constant time using O(n^3) processors. The details of this step are omitted.

7. Conclusions

Incremental graph algorithms deal with recomputing properties of graph after an incremental change has been made to the graph such as insertion of a vertex or an edge or deletion of an edge. In this paper we have described parallel algorithms for updating the distance matrix, shortest paths for all pairs and biconnected components of an undirected graph and topological ordering of the vertices of a DAG, after an incremental change has been made to the graph. Our algorithms require O(log n) time and use O(n^3) processors and therefore are efficient when compared to the start-over algorithms for initial computation of the above mentioned properties of graphs. We have shown that multiple inverted trees constitute a very useful data structure for developing incremental algorithms. It would be interesting to explore the applicability of this data structure to other graph problems.

References

[1] G. Cheston, "Incremental Algorithms in Graph Theory", TR 91, Dept. of Computer Science, Univ. of Toronto, Toronto (1976).

[2] E. D. Dekel, D. Nassimi and S. Sahni, "Parallel Matrix and Graph Algorithms", *SIAM J. Comput.*, 10 (1981), pp 657-675.

[3] S. Even and Y. Shiloach, "An On-line Edge Deletion Problem", *J. ACM*, 28 (1982), pp 1-4.

[4] S. Fujishige, "A Note on the Problem of Updating Shortest Paths", *Networks*, 11 (1981), pp 317-319.

[5] D. Hirschberg, "Parallel Algorithms for the Transitive Closure and the Connected Component Problems", *Proc. of Eighth ACM Symposium on Theory of Computing* (1976), pp 55-57.

[6] D. Hirschberg, A. K. Chandra and D. V. Sarwate, "Computing Connected Components on Parallel Computers", *Comm. ACM*, 22 (1979), pp 461-464.

[7] S. Pawagi and I. V. Ramakrishnan, "Parallel Updates of Graph Properties in Logarithmic Time", *Int. Conf. on Parallel Processing*, 1985 (to appear).

[8] C. Savage, "Parallel Algorithms for Some Graph Problems", TR-784, Dept. of Mathematics, Univ. of Illinois, Urbana (1977).

[9] C. Savage and J. Ja'Ja', "Fast Efficient Parallel Algorithms for Some Graph Problems", *SIAM J. Comp.*, 10 (1981), pp 682-691.

[10] P. Spira and A. Pan, "On Finding and Updating Spanning Trees and Shortest Paths", *SIAM J. Comp.*, 4 (1975), pp 375-380.

[11] Y. Tsin and F. Chin, "Efficient Parallel Algorithms for a Class of Graph Theoretic Problems", *SIAM J. Comp.*, 14 (1984), pp 580-599.

END

FILMED

4-86

DTIC